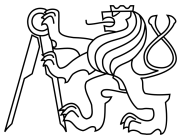
CZECH TECHNICAL UNIVERSITY IN PRAGUE
FACULTY OF INFORMATION TECHNOLOGY



ASSIGNMENT OF BACHELOR'S THESIS

Title:	Traffic connection visualization
Student:	Zdeněk Šlégl
Supervisor:	Ing. Jan Baier
Study Programme:	Informatics
Study Branch:	Computer Science
Department:	Department of Theoretical Computer Science
Validity:	Until the end of winter semester 2018/19

Instructions

Analyse the problem of finding traffic connections to a particular place from any point in the Czech Republic and choose an appropriate metric for computing a "traffic connection score" based on freely available data about bus and train traffic in the Czech Republic.

Based on the analysis, design and implement a desktop application that computes connection scores for user selected place. The sufficient level of detail for score computing is a district level (i.e., a score value for every district). Computed values should be displayable on the map, for example as a heat map. The application must allow to easily interchange the computing algorithm and/or the data source(s). The language for implementation will be C/C++, the application have to be compatible with Linux based systems.

Test the application properly and write a user manual.

References

[1] http://www.chaps.cz/files/cis/jdf-1.10.pdf
[2] https://www.mdcr.cz/Dokumenty/Verejna-doprava/Jizdni-rady,-kalendare-pro-jizdni-rady,-metodicke

doc. Ing. Jan Janoušek, Ph.D.
Head of Department

prof. Ing. Pavel Tvrdík, CSc.
Dean

Prague March 3, 2017

Czech Technical University in Prague
Faculty of Information Technology
Department of Theoretical Computer Science

Bachelor's thesis

Traffic connection visualization

Šlégl Zdeněk

Supervisor: Ing. Jan Baier

16th February 2018

Acknowledgements

On the first place I would like to thank to my supervisor for his patience. Thanks also goes to whole my family for their support and understanding during writing of this thesis.

Declaration

I hereby declare that the presented thesis is my own work and that I have cited all sources of information in accordance with the Guideline for adhering to ethical principles when elaborating an academic final thesis.

I acknowledge that my thesis is subject to the rights and obligations stipulated by the Act No. 121/2000 Coll., the Copyright Act, as amended, in particular that the Czech Technical University in Prague has the right to conclude a license agreement on the utilization of this thesis as school work under the provisions of Article 60(1) of the Act.

In Prague on 16th February 2018 .....

Czech Technical University in Prague
Faculty of Information Technology

Citation of this thesis

Šlégl, Zdeněk. *Traffic connection visualization.* Bachelor's thesis. Czech Technical University in Prague, Faculty of Information Technology, 2018.

Abstrakt

Cílem této práce je především vytvoření desktopové aplikace, která umožní najít nejlepší místo pro sraz při využití veřejné dopravy. První část práce se zabývá hledáním a analýzou dostupných zdrojů vstupních dat. V další části se zaměřuje na algoritmy řešící vyhledávání nejkratší cesty a jejich možnou aplikaci nač problém. Zkoumá také možnosti vizualizace dopravní dostupnosti. A konečně, v poslední časti, se za využití znalostí získaných předchozí analýzou, zabývá vytvořením aplikace. Ta je schopná zpracovat data, vypočítat dostupnost do všech okresů a na závěr tato data zobrazit na heat mapě.

Klíčová slova Vizualizace dopravní dostupnosti, předzpracování dat, veřejná doprava, desktopová aplikace, C++

Abstract

The purpose of this thesis is to create a desktop application helping people to find the best place to meet with use of the public transport. First part of the project examine different sources and formats of input data. In next part, main emphasis is on algorithms solving the shortest path problem and their possible application on our problem. Further, it examines possibilities of transport accessibility visualization. Finally, with the use of knowledge gained

during analysis, application is created. It is capable of collecting necessary data, calculating best connection to all tiers in Czechia and finally visualizing calculated results on a heat map.

Keywords Transport accesibility visualization, data preprocessing, public tranposrt, desktop application, C++

Contents

List of Figures

List of Tables

Introduction

After long monopoly of one company and several court trials lasting for years, Ministry of Transportation of Czech Republic have released public transport data in a machine-readable form. It opens hundreds of new possible ways on how to use them. In this work, I would like to examine these data and with the use of them, calculate transport accessibility to all tiers. After the calculation, they will be visualized. Best way of doing so will be analyzed.

My work should be useful for everyone, who ever wondered, what will be the best place to meet. Taking in consideration friends, coming from all parts of the country. The code, as well as the algorithms will be open to everyone and easily expandable. Therefore, allowing parties concerned, possibility for further expanding of my work. For example to add another countries or means of transport or more advanced search engine etc.

Because data suitable for machine processing, were released not long time ago, there is not many papers examining their possible benefits. Therefore, I believe, my paper could bring some new ideas. This is also main reason why I chose this topic.

Chapter 1

Analysis and design

1.1 Problem statement

Problem to solve is as follows. Several people from different parts of country wants to meet somewhere. Logically, everyone wants to travel for as shortest time as possible.

Outcome of this thesis should be desktop application, which will be able to process publicly available data about public transport in Czech Republic. Afterwards, by using suitable algorithm, it will calculate "traffic connection score" to all other tiers. Finally by using these results, application will be able to present these results in user-friendly way. Heat map will be chosen as the most suitable way of data visualization.

Because of diversity of each subproblem I will divide whole implementation process into three parts.

1. My task in the first phase is to preprocess big amount of public transport data, as find on the site of Ministry of Transportation in Czech Republic.

2. After processing relevant data, it will be necessary to find best algorithm and the way, problem could be represented. Problem is to find best traffic connection between tiers and to calculate average traffic connection for several people from several tiers.

3. Last step is to visualize results from previous step.

1.2 Data formats for public transport

To be able to analyze problem properly we need to have some general knowledge. In sections follow I will first take a look at data formats for representing public transport schedules. In next section we will sum up most commonly used definitions and algorithms. Last section of this part will describe possibilities of data visualization.

1.2.1 Public transport open data formats

In last few decades availability of open-source public transportation data underwent huge change. Nowadays most of the developed countries provide these data to everyone free of charge in easily readable formats. The most modern and widely spread one is indubitably GTFS. But as there are yet no legal standards, we can also found data in other formats. GTFS stands for 'General Transit Feed Specification' and as we can see at *Google Developer* [1] page, there are hundreds of cities worldwide involved. We can find here ones from US, Canada, England, but also Finland, Estonia, Spain etc. This format was invented by a small group of transit agencies in the USA. The need for electronic form of transportation network and schedule data, led to invention of GTFS format. Quickly, it became an industry standard for publishing public transport data. [2] Nowadays variety of third-party software applications use these data. Most commonly they are used for trip planning, creating of timetables, data visualization and many more. The reason GTFS stands out is it's relative simpleness to create and read data by people as well by machines. [3]

A GTFS format consist of minimum 6, but up to 13 CSV(comma-separated values) files. More detailed structure of these data could be seen in the figure 1.1.

1.2.2 Situation in Czechia

The idea of open data in state-owned companies doesn't have long tradition in Czech Republic. When talking about public transportation, for finding best connection, there had been basically monopole of one company, Chaps s.r.o. All companies in the country, who provide public transport service, are obliged by law to deliver data about their connection to Chaps. But for many years Chaps didn't have to provide these complete data to anyone else. Basically only IDOS, owned by Mafra a.s. received the complete data. This symbiosis was very favorable for both companies. Only one server in the country, providing transport planning, logically attract to many visitors to their site. Which in the end brings them more money through advertisement etc. There had been several companies trying to broke this monopoly. In the beginning Chaps was willing to provide their data, but only for quite high prices. Later he stopped providing data, to everyone except IDOS, altogether. In the end it was only Seznam.cz, who never gave up and still try to get the data. They sue Chaps for not providing machine readable data and after many years Chaps provided these data to public. Currently we can find all of public transport schedules available at *Ministry of transportation* [4]. Bus schedules are provided in JDF format. Which is similarly to GTFS represented by several CSV files. Meanwhile train schedules are in Kango format. More detailed elaboration about these data formats could be find in section Format

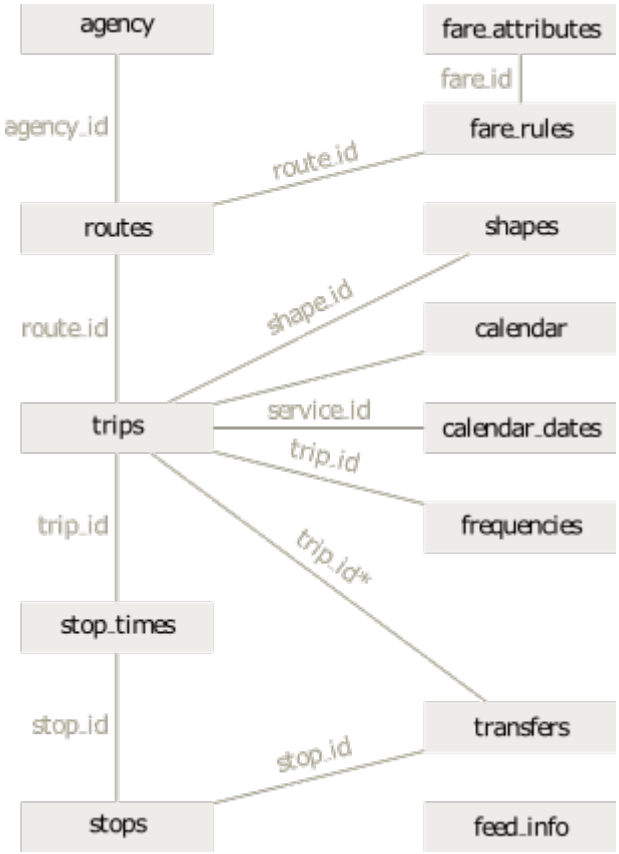


Figure 1.1: GTFS diagram

of available data. Also nowadays some cities, like Prague or Pilsen, provide data in GTFS format.

1.3 Best connection

So let's see what a problem is. We have several stops around Czechia. Our user comes from one of the places. Our goal is to calculate "traffic connection score" to all other stops. Apparently to calculate score, we are finding shortest path between these points. This suggest problem will be graph assessment. We have stops which represent graph vertices and connection between them - graph edge. Time needed to cover the distance would be weight of edge. Cause bus or train could and will go in both direction and there might be several connections between each pair of stops, we know we will be dealing with directed multi-graph. Problem appears, when we realize both nodes and vertices having several additional parameters. On this place I would like to mention definitions and algorithms used in graphs.

1.3.1 Graph theory

There is no to make my own definitions, so I will use one from Encyclopedic Dictionary of Mathematics.

In the most common sense of the term, a graph is an ordered pair G = (V, E) comprising a set V of vertices or nodes or points together with a set E of edges or arcs or lines, which are 2-element subsets of V (i.e. an edge is associated with two vertices, and that association takes the form of the unordered pair comprising those two vertices). To avoid ambiguity, this type of graph may be described precisely as undirected and simple.[5]

1.3.2 Shortest path problem

In this problem, we are trying to find the shortest path from one node or vertice to another one. Shortest path is defined as one where sum of weights of its edges is minimized. Weight could be represented by many different ways. In representation of real life map it could represent distance between nodes or time needed to cover it, depending if minimalizing of distance or speed is our purpose.

There is several different variations of shortest path problem, distinguished by how many paths we want to calculate. Simplest one is when we have two nodes and we only want to calculate distance between them. This one is called single-pair shortest path problem. If we have one source node and are calculating distance to all other nodes, than we are talking about single-source shortest path problem. Last but not least is situation, where we need to calculated distances between all possible combinations of nodes. In that case we call it all-pairs shortest path problem.

1.3.2.1 Dijkstra's algorithm

Very popular algorithm which solves shortest path problem. In particular it solves its single-source version where all edges weight are positive values. How does this algorithm works could be seen in pseudo-code below.

For me, the algorithm is not useful in this form. We will be able to use it for example if we will be searching for shortest path on road map. Defect of basic version of Dijkstra's algorithm when trying to calculate public transport connection is, that it expect firmly set value of edge. In our case, we have to take into consideration starting time from node, which differs every time and also waiting time in node between two connections. That doesn't mean we should reject this approach altogether. We will take advantage of main idea, but modify it.

Algorithm 1 Dijkstra's algorithm

```
1:  function DIJKSTRA(G : Graph, s : source)
2:      create vertex set Q
3:      for each vertex v in Graph do
4:          dist[v] ← ∞
5:          parent[v] ← NILL
6:          add v to Q
7:      dist[s] ← 0
8:      while Q ≠ ∅ do
9:          u ← EXTRACTMIN(Q)
10:         remove u from Q
11:         for each neighbor v of u do
12:             if dist[v] > dist[u] + weight[e] then
13:                 dist[v] ← dist[u] + weight[e]
14:                 parent[v] ← u
        return dist[], parent[]
```

1.3.2.2 Floyd–Warshall algorithm

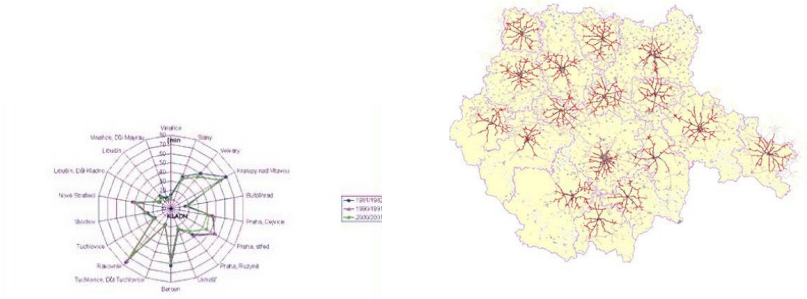
Let's also take a look at Floyd-Warshall algorithm. Main advantages of this algorithm compared with Dijkstra's one is it's simplicity to code. What's even more important, it calculates not only shortest path from one node to all others, but even shortest path between all pairs of nodes.

Pseudo-code below explains how algorithm works.

Algorithm 2 Floyd–Warshall algorithm

```
1:  let dist be a |V|x|V|array of minimum distances initialized to infinity
2:  for each vertex v do
3:      dist[u][v] ← 0
4:  for each edge (u, v) do
5:      dist[u][v] ← w(u, v)                    ▷ the weight of the edge (u,v)
6:  for k ← 1, V do
7:      for i ← 1, V do
8:          for j ← 1, V do
9:              if dist[i][j] > dist[i][k] + dist[k][j] then
10:                 dist[i][j] ← dist[i][k] + dist[k][j]
```

Unfortunately, even this algorithm is not suitable for us, or at least not without further editing. Basically there are same problems as with previous one. It couldn't take into consideration waiting time between connection. What's more it couldn't handle varied results for different arrival times to node.



(a) Accesibility change over time (b) Road accesibility

Figure 1.2: Graph visualization examples

1.4 Possibilities of data visualization

Third and last theoretical section will be all about visualization of data. Let's look on some possibilities how to visualize transportation accessibility map.

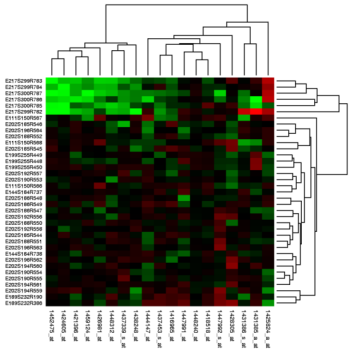
Data visualization underwent huge change in last decade or so. Everything is faster, amount of information is growing, so need to explain complex problems as simply as possible is logical. Visualization of data is great way how to achieve that. You can present huge amount of data on just few graph and what's more, people understand what the data says.

Visualization of transport accessibility is only small part of area, but there is still variety of application and ways how to do that. Two main possibilities on how to calculate transport accessibility are time accessibility and distance. It is evident, that each stakeholder will have different interest, so that different calculation will be preferred. Carriers for example care more for distance as they have to pay for oil and tolls. On the other hand people traveling by public transport doesn't really care, how many kilometers will the route covers, but they will care more how much time it will take. In this work, I will focus mostly on time accessibility.

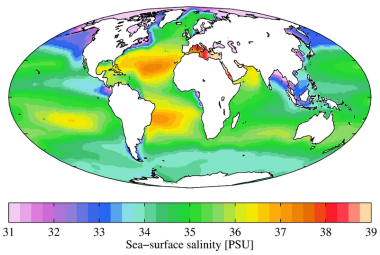
One of the ways how transport accessibility could be visualized is graph visualization. The first example in Figure 1.2 show transport accessibility from Kladno to villages around it. Use of this type of visualization allows to show three different results from three different years. Second example shows accessibility with different width of line.

1.4.1 Heat map

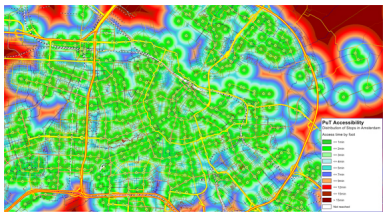
Heat map is definitely solution, which dominate all others. What it actually is a heat map? Basically it is every graphical representation of data, where each value is represented as a color. Term "heat map" was actually trademarked



(a) Gene in several conditions



(b) World salinity



(c) Stops accesibility

Figure 1.3: Examples of heat maps

in 1991 by Cormac Kinney, software designer, but luckily only till 2003, when Kinney's trademark was unintentionally canceled.

There is many types of heat maps. For example, tree map, mosaic plot or a density function visualization etc. We are interested the most in a version, where there is usually some background picture and then each point on a picture have some value. According to how big or small value is, relative to others, the color will be chosen.

Also we have different color schemes. Most commonly used version is rainbow color-map. It have many versions, depending on how many colors one include in rainbow. We can also use gray-scale color-map. But its disadvantage is, that humans can perceive less shades of gray than shades of color. Few examples could be find at Figure 1.3.[6]

1.5 Existing solutions of a problem

Even after long analysis of already existing solutions, I didn't came to program, which will be solving same issue as stated. However, there is considerable number of projects, which solve smaller or bigger parts of the problem.

As it could be deducted from previous sections, biggest issue, stopping people from solving similar task is the format of data provided by Chaps. There are several projects able to show area accessible in set time period from

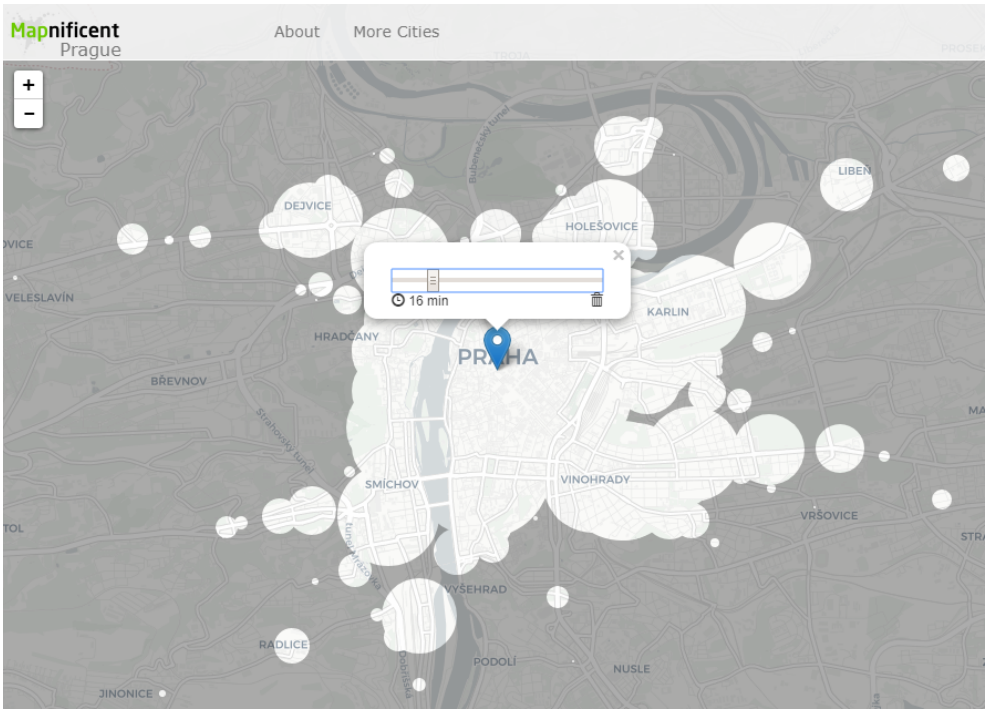


Figure 1.4: Mapnificent application

any location. One thing they have all in common is request for input data to be in GTFS format.

To name few, there is web application called *Mapnificent* [7]. In this free software you can just pinpoint any place on map. Set time period and you will get map showing where you can get to during this time. In this project, we can find several cities from USA as well as Europe. Because Prague already provided data in suitable format, we can even try this app in Prague.1.4. As it is free software, everyone who have some data, could add them to the application and it will show you similar map of your area.

Another project, thought not free, is Trulia Maps. Advantage of this application compared with Mapnificent is, that it not only shows which areas you can reach in set time period, but also distinguish, using heat-map, how long it will take to get to each place. On the other hand, disadvantage is, it is not possible to add user data. Example of map of New York showed in Figure 1.5

Both projects works similarly. They preprocess the data, then they calculate the values for each place using modified Dijkstra's algorithm, only to show the values in map at the end.

As we can see the way of calculating values in my project could be very similar as the one used in above mentioned projects. Difficulties and difference become when we will try visualize data. GTFS contains exact GPS coordinates

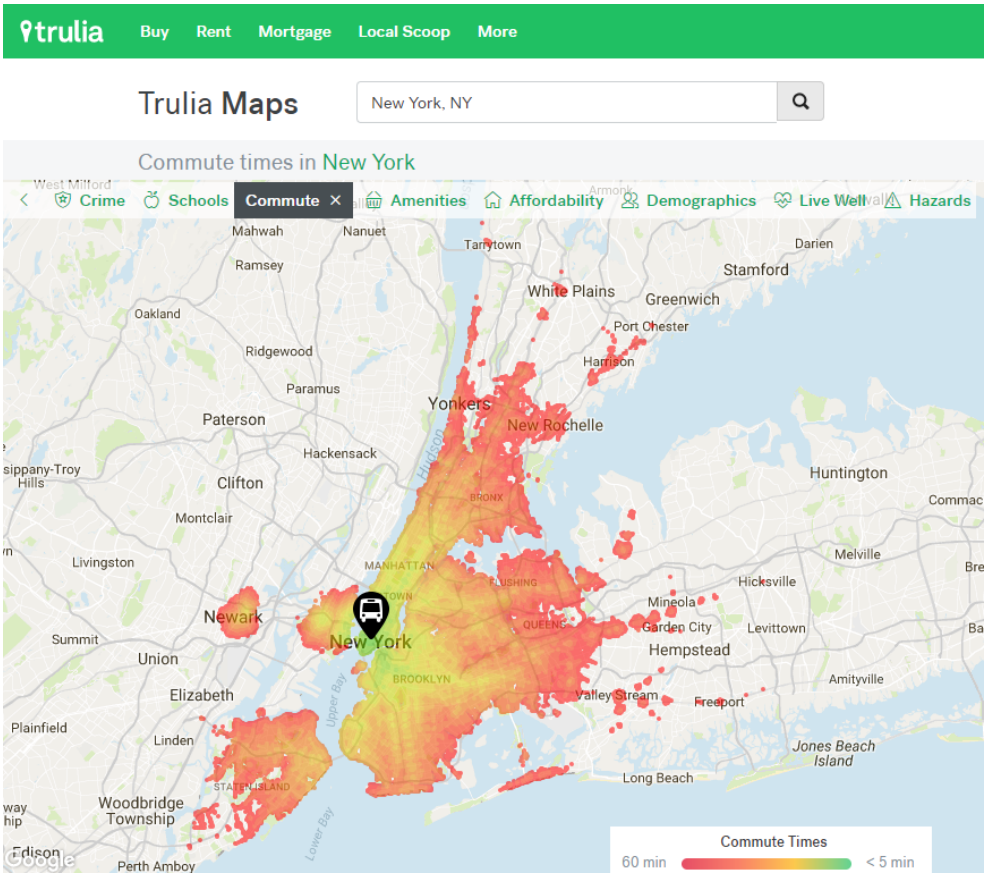


Figure 1.5: Trulia Maps application

for each stop. That means that with using either Google maps or Openstreet maps, we can easily visualize these data using either of map base.

And it is also biggest problem. Even if we will somehow manage to process JDF and Kango format, trying to format them as GTFS, there will be several informations missing. The most important one missing are GPS coordinates of stops. As a result there is no way we can project these data in map automatically.

1.6 Application design

In this section, I would like to show, how my application is build and how it works. This will be demonstrated on component and class model.

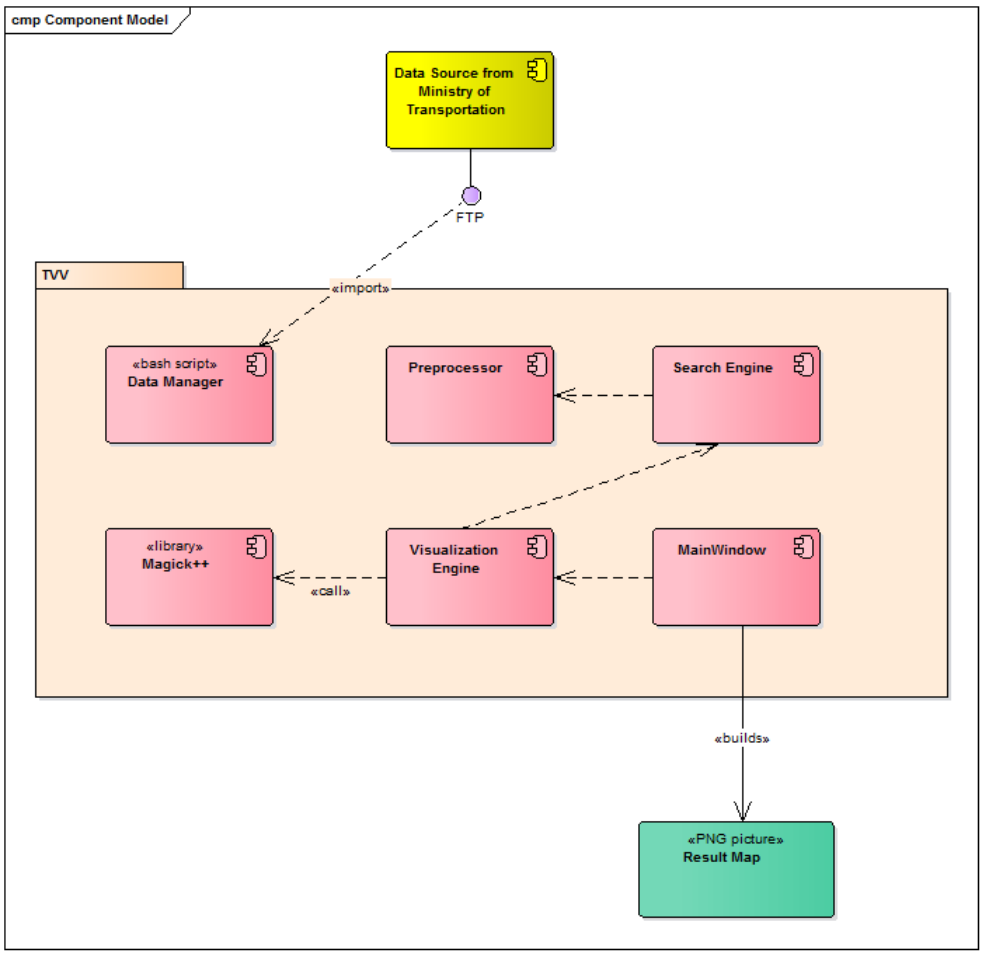


Figure 1.6: Component model

1.6.1 Component model

Which components does my program consists of and how they interact could be seen best in Figure 1.6. There are 4 most important components, who interact with three others.

- **Data Manager** - This component takes care of downloading data from FTP server of Ministry of Transportation and transforming them to more friendly format afterwards.
- **Preprocessor** - His main duties are to use data prepared by *Data Manager* and to preprocess them. During this process component disqualify invalid data and transform problem to smaller scale by selecting only relevant data. Results are saved to a file in a format, understandable by *Search Engine*.

- **Search Engine** - Core part of our application. Uses data prepared by *Preprocessor* and applicate an algorithm on them. After results are calculated, it provides them to *Visualization Engine*

- **Visualization Engine** - Component who takes care of data visualization. Gets data from *Search Engine* and by using *Magick++* library, builds a *Result Map*.

- **MainWindow** - Component taking care about GUI and communication between front end and back end.

- **Data Source from Ministry of Transportation** - A FTP server which contains all necessary input data.

- **Magick++** - API to the ImageMagick library enabling *Visualization Engine* to build *Result Map*.

- **Result Map** - Heat map builded by *Visualization Engine*.

1.6.2 Class model

My program consist of 4 classes. In the Figure 1.7 you can find model of all classes and relations amongst them. Graph, represents data model for imported data. This class contains main search logic and shortest path algorithm. Graph is created by several Nodes and Edges. Node represents one stop in our map. Edge is a single connection between any two stops in a map. Last class Visualize is here draw a results into map.

1.7 Programming language

For the solution of this problem I chose C++ programming language. Some parts like visualization of the results are not exactly suitable for object oriented approach. I didn't want, however, to mix it with pure C. On the other hand Graph representation is copybook example of using OOP. To receive data from FTP server and to do some basic preprocessing I used Bash scripts with occasional use of AWK.

1.8 Format of available data

As I already mentioned above we have two different data formats for each bus and train transport.

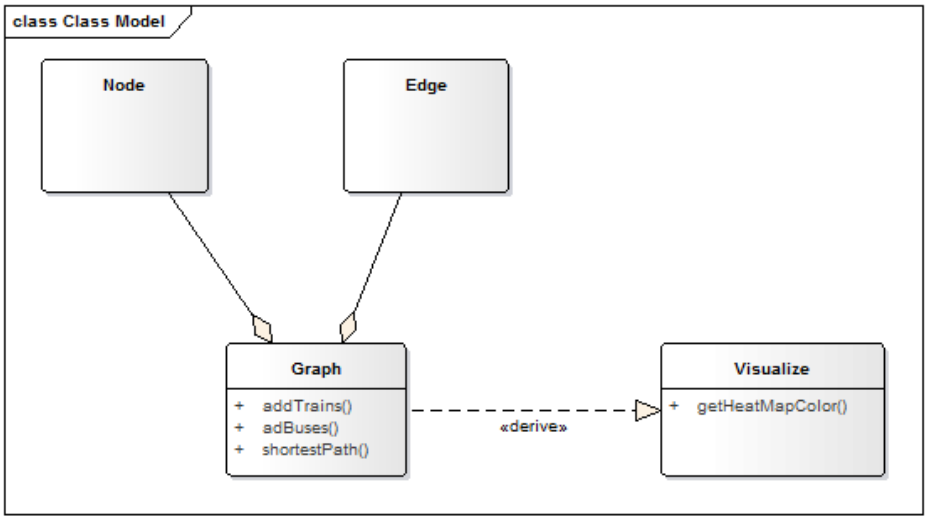


Figure 1.7: Class model

1.8.1 Bus Transport

All bus lines are provided in one ZIP file. After extracting it, there is more than 8 thousand ziped files. Each for one line as provided to Chaps by other carriers. In each folder we will find nine up to seventeen files following standards of JDF (Jednotný datový formát). Detailed specification could be find at *CHAPS* [8] This is basically several CSV files standardized by Ministry od Transportation in Czech Republic.

Direction tree of JDF zip file looks as follows. Files listed here in each folder are only the mandatory ones. Each file represents one line as in Figure 1.8

The most important one is definitely file *ZASSPOJE.TXT*, which contains informations about at what time at which stop particular line will stop.

Listing 1.1: ZASSPOJE.TXT example

```
"690310","7","19","17","","","","","","","15","","0731","","","1";
"690310","7","20","16","","","","","","","","<","<","<","<","1";
"690310","7","21","18","","","","","","","15","0734","","","","1";
"690310","9","1","4","","","","","","0","","0835","","","1";
```

As we can see, this file contain one line for each stop and each connection. Unicity is assured by line number and its specification, connection number and stop tariff number. As a result, stops of connections are always sorted

Figure 1.8: Line folder contents

```
1..8400.zip ..... zipped file for each line
├── CASKODY.TXT ..... Specifications of line atd days when it goes
├── DOPRAVCI.TXT ..... List of carriers on the line and their informations
├── LINKY.TXT ..... Specification of line
├── PEVNYKOD.TXT ..... List of special atrributes of line
├── SPOJE.TXT ..... List of all rides
├── VERZEJDF.TXT ..... Version of JDF
├── ZASLINKY.TXT ..... Stops on a particular line
├── ZASSPOJE.TXT ..... All stops and their times
└── ZASTAVKY.TXT ..... List of stops
```

Table 1.1: ZASSPOJE.TXT columns description

Line number	Six figures line identification with relation to LINKY.TXT
Connection number	Odd in one direction, even in the opposite one.
Tariff number	Increasing number for each connection
Stop number	Number of stop in this line, relation to ZASTAVKY.TXT
Mark code	Special mark with relation to OZNACNIKY.TXT
Platform number number	Not mandatory
Column 7-9	Codes for special attributes of line
Distance	Distance from start of connection in kilometers
Arrival time	Mandatory in final stop
Departuture time	Optional in final stop
Column 13-14	Not used in public transport
Line specification	Distinguish if several possibilities of line, relation to LINKY.TXT

according to line stops. So for return ride time and distance values are in reverse order.

For each stop connection, departure time is have to be set. Exceptions are when connection is only passing through stop. In that case departure time value is "|". In case connection is going different direction, value is "<".

In case connection stays in any stop for more than 5 minutes, also arrival time have to be stated.

Detailed description of each column, from first to last can be find in table 1.1.

The most important for us will be line connection number, to distinguish different connections, stop number and departure and arrival times.

Table 1.2: LINKY.TXT forth column description

A	City line
B	City line serving suburbs
N	International line without intrastate transport
P	International line with intrastate transport
V	Domestic line within region
Z	Domestic line - interregional
D	Long distance domestic line

Another important file is ZASTAVKY.TXT. It contains informations about stops of particular connection.

Listing 1.2: ZASTAVKY.TXT example

```
"20","Prostejov","","aut.st.","","","19","28","","","","","";
"21","Prostejov","","Ujezd","","","19","","","","","","";
"22","Pustimer","","","","","","","","","","","";
"23","Sumperk","","aut.st.","","","19","","","","","","";
```

First column contains stop number. Another three information about stop area, village name and stop name. Next columns provide informations ID of carrier and its country. The rest of them are Mark codes.

Last file I will use, is LINKY.TXT. In this file in forth column I will find "Type of line". It tells us whether certain line goes only inside of the city limits, in whole region, country or if it's international one. Which characters could be used and what they represent could be seen in table 1.2.

To make things more complicated there is several versions of JDF. Format I presented above is one of version 1.11. But amongst folders we can find several different versions, included invalid ones like 1.8. Differences between them are not minor. For example version 1.10 has one less column in ZASSPOJE.TXT and 1.8 has even two column less than 1.11. Also there are few files which are not valid all together.

1.8.2 Train transport

Even-though JDF format seems unknown, it is at east somewhat similar to internationally recognized GTFS. But Kango format, which is used for train data isn't even closer. After unzipping file called VS_2018.zip we find 27 files with same name, but very creative extensions. All important data could be find in one huge file called *JizdniRad2017.trv*. As we can find in documentation trv stands for Train route("trasa vlaku" in Czech language).

Listing 1.3: JizdniRad2017.trv example

```
70/0|0054|500306|00|70|||0||||||0|11|25|||1||...||
70/0|0054|583765|00|||1|0|11|27|||||||||||||...||
71/1|0054|583765|00|71|||||||||0|6|28||13|1||...||
71/1|0054|500306|00|71|||0||||||0|6|30|||1||...||
```

There are tons of possible attributes, but vast majority of them is usually not set. Important informations for us is first value, which represents train number. Third column sets stop ID. Another necessary informations are arrival and departure hour and minute. First two could be find ninth and tenth column, other pair in fifteenth and sixteenth one.

Second necessary file is *JizdniRad2017.db* which contains informations about stops. Most importantly it maps stop ID to real stop name. In other files we can find tons of informations about carriers, locomotive etc., but they are not useful for our purpose, so that I will not examine them further. For fans of train, detailed documentation could be find at *FTP server of Ministry of Transport* [9].

1.9 Data preprocessing

1.9.1 Taking only relevant data

To prepare data to the form, more suitable for my problem, I did several steps. From the user perspective all these steps are done by running *configure.h* bash script. It download both train and bus data, unzip all folders and delete unnecessary ones. Next problem is coding. Chaps provide all data in windows-1250 coding, but for Linux we need utf-8. So that I have to recode all files.

As it's obvious from a description of the formats, there are tons of information which are irrelevant for my purpose. Therefore I will take only ones, which will be useful.

1.9.2 Decreasing number of stops

Keeping in mind, available data doesn't provide any informations about their location, I decided to dramatically decrees number of stops considered in my graph. Why the lacking of GPS coordinates are such a problem? Well let's say, I calculated score to every single stop. So now I have one source stop and hundreds of others with calculated time needed to reach them. But what now? How to visualize them?

I will have to manually search for GPS coordinates for every single stop. I think everyone will acknowledge that it will be really boring and time consuming. So what I will probably do? Choose only some number of main stops with high data validity. But how many stops to choose? What about to take only capital cities of each region. Well there is only fourteen of them and

if there will be people living close by, my problem will not be very suitable. Especially historically, Czechia was divided into 76 tiers plus Prague. This amount of stops seems big enough to precisely show transport availability and is still only reasonably time consuming to set their location.

So in the end, we will only use these 77 stops and dismiss the rest. Further, I will work only with these nodes already during creating of my graph.

1.10 Searching a place to meet

1.10.1 Creating the graph

First step have to be data import and to create correct graph. This process will contain two steps. In the first one I will add bus data, in the second one, train data.

Creating nodes is super simple. We already chosen 77 stops which will be represented by 77 nodes. Each node will be stored as Node class. Each node contain information about it's ID and sets of outgoing and incoming Edges.

On the other hand creating of Edges which connects them is much more complicated. Class Edge will represent single connection between two Nodes (stops). It will holds informations about its departure time from origin and arrival time do the destination stop. Also it have to remember what its origin and final Nodes are.

We have to realize, that when adding connection which goes for example between 5 relevant stops, we will have to add it 4 different Edges between 5 Nodes. For example let's look at line number "7312".

Listing 1.4: Time schedule of line 7312 example

```
Liberec,,aut.nadr.,"","1330";
Mlada Boleslav,,aut.st.,"1420","1425";
Praha,,UAN Florenc,"","1520";
Kutna Hora,,aut.st.,"1620","1625";
Hradec Kralove,,Terminal HD,"1725","";
```

This line starts in Liberec and then continues through Mlada Boleslav, Praha, Kutna Hora to Hradec Kralove. Therefore. this example creates four different edges. First will be between Liberec and Mlada Boleslav. Departure time from origin being 13:30 and arrival time to final destination 14:20. Second between Mlada Boleslav and Praha. Departure time 14:25, arrival 15:20. etc.

With this example I am trying to demonstrate fact, we couldn't automatically take arrival time as time, when connection gets to the stop. We always have to check if arrival time is present.

Furthermore bus can go in opposite direction. That means stop which came as first one could also be departure one. So we always have to check which one is which.

1.10.2 Finding the shortest path

When our graph is finally ready and filled with all the data we can go forward to next step - finding shortest path between source stop and all other stops. After inspecting several known algorithms I decided to choose Dijkstra's one as a base. Main idea will remain the same, however I will have to edit it quite extensively. There is several things basic Dijkstra's version is unable to process. Firstly, we are working with some starting time set by user, so that first node will already have this value. Secondly, what's more complicated to implement, is fact that there might be some waiting time between nodes and it's length depends on time of arrival.

Let's imagine a stop, say Brno for example. We will travel by bus to this stop with intention to continue to Breclav afterwards. Brno will be final destination of the bus, so we have to get out of bus and wait for our connection to Breclav. According to schedule next bus will leave in twenty minutes and then will take one hour to get there. Connection to Breclav represents Edge and its value is 60. But the real value of this connection will be 80 (20 waiting time + 60 ride time). But what if we miss first bus to Brno and arrive two hours later and for bus to Breclav we will have to wait forty minutes. Than, assuming ride time will be the same, this same route will have real value of 100.

Another problem which basic algorithm is unable to solve is a situation when there is a bus leaving from our stop in 30 and another in 40 minutes. We couldn't just say the way that leaves first is the best, because first one could be slow regional bus taking one hour for the ride, later could cover in only 30 minutes. Logically second will arrive earlier so that we want to use that one.

1.10.3 Meeting point

When we are trying to decide which place to choose for meeting, we have to set our priorities. For example if there will several people living close by and only one somewhere far away, will we try to meet somewhere half way, so that everyone have to travel for same amount of time? Even-though this might seem fair, it is definitely not very efficient for many people to travel quite some distance only because of one person. I decided to calculate it in a way, so that overall travel time of everyone is minimal. Easiest way how to secure this, is computing travel time to all other places separately for each person. Than calculating average travel time to each tier. I think it is the most fair option.

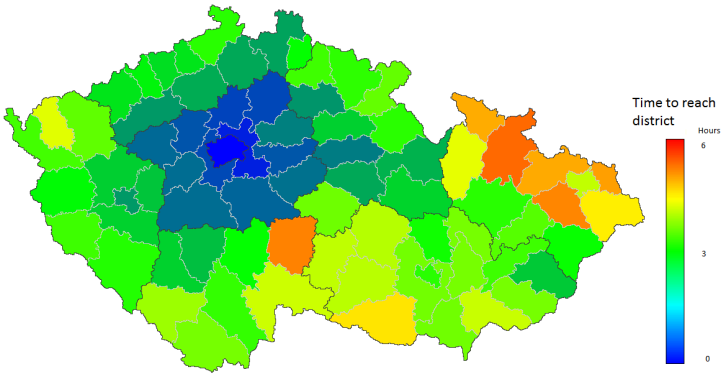
1.11 Visualization

There is many ways how my results could be visualized. I analyzed several options of graph visualization etc., only to came back to first idea which came on my mind - heat-map. I chose this option, because I think it is the most clear way how to present results. You don't have to know much and still understand what it shows. Now when I chose heat-map, I have to chose specific technical solution. There is several ready to use solutions, using Google or OpenStreet maps, but problem of all of them, is fact, they require GPS coordinates, which I don't have. After some research I came to nice solution. I could get picture of Czechia with tiers as a silhouette. And than, flood fill each tier with color according to its value. There is several possible heat-map gradients. You can choose only black and white color or up to seven different colors and even more. I decided to go for 5 colors as a good compromise. Then I had to find some application, which will allow me to calculate exact colors and them paint it. Best suiting solution I find was *ImageMagick Magick++ API*. This API is really simple, but powerful enough to solve everything I need.

1.12 Graphical user interface

For easier communication between user and back-end I created simple GUI. For this purpose I am using Qt GUI module [10]. From the list on the left user can choose several places, from whose people are from. On the clock he should set departure time and then only click button "Calculate". Heat map with calculated accessibility is calculated and shown afterwards. Picture of my GUI could be find in the last chapter.

(a) Silhouette only



(b) Heat map

Figure 1.9: Changing silhouette to heat-map

Chapter 2

Realization

2.1 Preprocessing

2.1.1 Taking only relevant data

Let's first start with the buses following the rule to Eat a frog first. Each line, except of several others, contain file called LINKY.TXT. Forth column tells us operational area of the bus as in table 1.2. Because we are working only within Czechia all lines with letter "N" could be easily dismissed, as they don't allow passengers to travel domestically. As we decided to work only with main stops of each tier, we can also skip lines of type "M", because they operate only inside cities.

In next step file called *easySpoje* is created with use of bash scripts. As an input this script takes file ZASSPOJE.TXT and select only columns which contains local stop code, code of particular connection, arrival and departure time. Empty time values are replaced by "-1".

We couldn't forget trains have completely different format, so we have to preprocess them separately. From the file *JizdniRad2017.trv* I will only need first column - train number, third column - to get code of the stop and then ninth-tenth for arrival times and fifteenth-sixteenth to get departure times. Using bash programming language awk, I will cut these columns and insert them to separate file. Replace again empty values with "-1" to make working with them later easier. This file, which I called *Rad-clean*, than looks like below.

Listing 2.1: Rad-clean example

```
556803 5527/7 12 49 12 49
531202 5527/7 12 53 -1 -1
531202 5528/8 -1 -1 17 4
556803 5528/8 17 7 17 7
556613 5528/8 -1 -1 17 8
```

2.1.2 Decreasing number of stops

Problem here, we are trying to find an answer for, is whether are we able somehow automatically recognize, which stop is the main one for each tier. My answer is that no, or at least not with my level of knowledge. Method I used is as follows. I gather a list of all tiers and their capital cities. Then search for biggest stops in both bus and trains transport. The result are two list called *relevant-stops*. One for trains, which contains stop code number. Another one for buses. Problem with main bus stops is the fact, in bigger cities, there are several places, where buses are leaving from. This issue was most visible for Prague, where there are around 8 places, buses are leaving from. I did quite a simplification when I take all these stops as one. But because, my task is not to make schedule planner, but traffic accessibility map, this simplification will not make a big difference.

Furthermore for several stops, there have been differences with exact name, even thought it describe same stop. Good example could be Brno tier, whose main stops is written in three different ways. Sometimes it even didn't follow what is region and what local place.

Listing 2.2: Differences in naming convention

```
Brno , ,UAN Zvonarka ;
Brno [BM] , ,UAN Zvonarka ;
Brno ,UAN Zvonarka , ;
```

Already in main C++ program last preprocessing is done. This was too complicated process to do only with bash scripts. For this purpose there is function called *createGoodLines*. What it does is taking one line by one and checking if there are at least two stops from the list of relevant-stops. If yes than it saves number of line, together with pair of local stop code and global stop code. Reason why I have to do this step is the not really clever way of JDF format store data. There is no global unique code for each stop. Each line have their own stop codes numbered from 1 and increasing. That means there are thousands of stops with same number "1". So I search each stop name in my list of *relevant-stops* and if I find it there I add to the local stop code also global number of tier. Tiers codes could have values from 1 to 77 as there is 76 tiers plus Prague. Numbering is set by region of each tier and by alphabetic order in each region.

Listing 2.3: goodLines file example

```
5283 4 54 8 69 20 72 22 60 29 76 33 63
5285 3 58 6 1 7 1 10 61
5289 21 27 24 12 25 12
5293 1 54 2 57
```

2.2 Best place to meet

2.2.1 Creating graph

Creating a graph basically means creating of one instance of class Graph. This class is holding a vector of Nodes, which holds list of Edges connected to each Node, so that I can access whole Graph. To create Graph we call constructor with user defined start time as parameter. Inside constructor we also create all Nodes and add them to the vector of Nodes.

Next step is adding Edges to the concerned Nodes. This work is done in methods of Graph *addBuses* and *addTrains.* In *addBuses* method I load *goodLines* file and then go one line in a time and open lines which are in this file. For each line I load *easyConnection* file. After, I check each stop one by one and if bus makes a stop there and also it is one of main stop of tier, I call *addEdge* method.

Similarly for trains I load file *Rad-clean* together with relevant train stops. Then I check one stop by one and if it is valid one I call *addEdge*.

Method *addEdge* first checks, whether this stop number is relevant. Next step is validation that both stops are in same connection and that origin is not the same. Finally it have to find out, if connection leads from first stop to the end or vice versa. This could be decided by evaluating connection value. If it is odd number than it goes from first one, even number means opposite direction. When all variables are set at proper value, we can finally create new Edge and add it to the set stored in concerned Node.

2.2.2 Finding the shortest path

All the logic is implemented in Graph class' method *shortestPath*. Only input variable is ID of source stop. The algorithm is inspired by Dijkstra's algorithm with necessary modification. First step is to create an array of distances from source for each Node and initialize them with big enough number. Then I insert one invalid Edge to the set of outgoing Edges, so that we are not out of bounds at the end of algorithm. Overloading of operator "<" secure, Edges are sorted by departure stop ID and, if same, than by departure time. Initialization of priority queue is also done. Pair in queue are two integers, with first being distance and second ID of Node. Last thing we have to do, before continuing to main part of algorithm, is to set distance to origin stop as user defined starting time and add it to the queue.

Listing 2.4: shortestPath method initialization

```
int *distance = new int[STOPNUMBER];
for (int i = 0; i < STOPNUMBER; i++) {
  distance[i] = BIGNUM;
  list[i].getDepartures().insert(Edge(-1, -1, 999, 999));
}

priority_queue < pair<int, int>, vector < pair<int, int >>,
    greater < pair<int, int >> > queue;
queue.push(make_pair(this->startTime, source)); // origin Node
    have distance of start time defined by user
distance[source] = this->startTime;
```

After we successfully initialized and set distances and queue we can go on into main loop of queue. First we find out which Node we are at and pop it out from queue. Than we load all outgoing connections, which we go one by one finding which will be earliest at destination stop.

Listing 2.5: shortestPath method queue loop

```
while (!queue.empty()) {
  int orig = queue.top().second;
  Node dest = list[orig];
  set <Edge> out = dest.getDepartures();
  int prevDest = -1;
  int minWeight = BIGNUM;
  for (auto num : out) {
    // going through all outgoing Edges finding the one with
        earliest arrival to next Node
    ...

    prevDest = num.getDest();
  }
}
```

Inside for loop, we take each outgoing Edge one by one and if this node was not calculated already, we check if there is shorter way to Node through others Nodes. If yes, we change distance and insert previous Node back to queue.

Listing 2.6: shortestPath method checking distance

```
if (prevDest != num.getDest() && prevDest != -1) {
  if (distance[prevDest] > distance[orig] + minWeight) {
    distance[prevDest] = distance[orig] + minWeight;
    queue.push(make_pair(distance[prevDest], prevDest));
  }
  minWeight = BIGNUM;
}
```

Still taking outgoing Edges one by one. If departure time of Edge is same or later than value of the node, we check if this connection is not faster than

the previous one. If yes, we set this as a new value to minWeight.

Listing 2.7: shortestPath method best connection between same nodes

```
if (distance[orig] <= num.getDepTime()) {
  int weight = num.getArrTime() - distance[orig];
  minWeight = weight < minWeight ? weight : minWeight;
}
```

2.3 Visualization

To visualize results in a map, I have created class Visualize. I decided to use ImageMagick Magick++ API as it's simple, but good enough for my purpose. This class basically only have Constructor, which gets array of int with final times as an parameter and one method called *getHeatMapColor*. Thy way it works is really simple. First it loads map with silhouette of Czechia together with coordinates of each tier as calculated by *shortestPath* method. Then it goes tier by tier and for each value of time accessibility it calculates color and then fill the tier with that color. At the end it saves image as *result.png*. Function to calculate the color was inspired by Andrew Noske's site [11].

Chapter 3

Testing

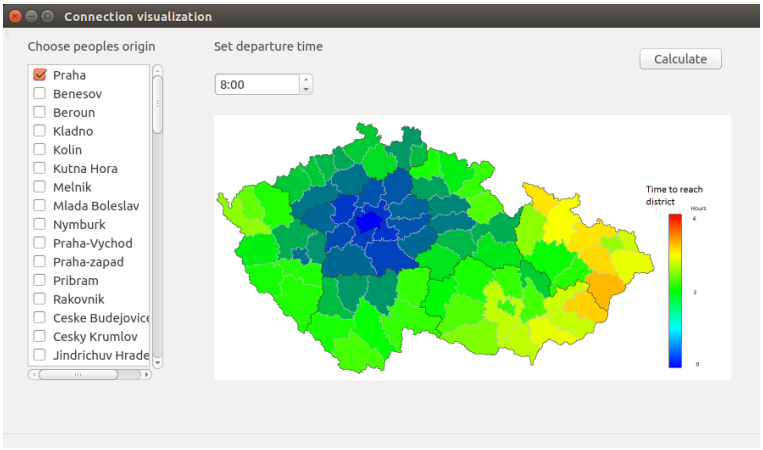
In this chapter I will test my program. For quite long time I was deciding how to easily validate the results. In the end, I will always check connection to three places on IDOS. There could be theoretically mistake in other places, but probability is very low.

3.1 One source

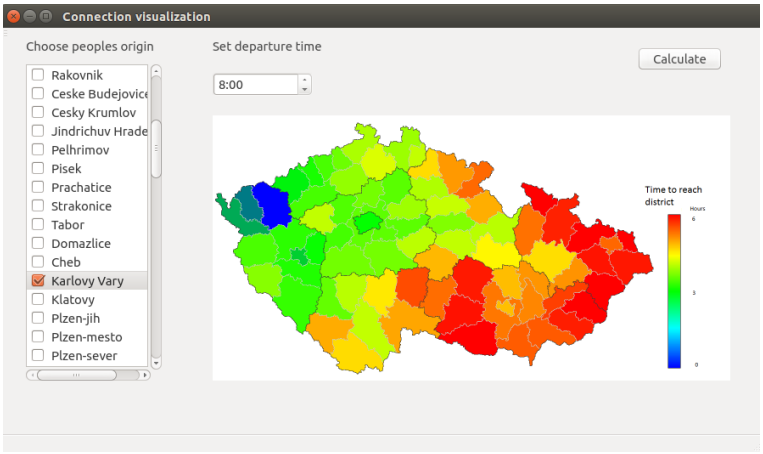
To start with let's choose Prague as a source. Starting time would be set at 8 am. So lets see node 17 Pelhrimov. Departure time from Prague at 8:20, transfer at Humpolec and arrival to Pelhrimov at 10:40. That means 2 hours and 40 minutes. From the map, we can see approximate time, which seems similar. To be sure we can check exact times in terminal. And it's exactly same. Second stop lets chose Sumperk. IDOS say departure 9:10 and arrival 11:26. Our program says 3 hours 26 minutes. And last one being Bruntal. According to IDOS arrival time 12:23. My program says 4 hours and 23 minutes. As we can see, for the first example we were able to find exactly same connection.

Next source I chose are Karlovy Vary. I decide to chose same three destinations so that we can validate data for more sources later. First ride to Pelhrimov. IDOS says arrival at 14:00, program agrees. Sumperk - IDOS - arrival time 13:41, my program answers 5 hours 41 minutes. Bruntal - IDOS says arrival 15:53, but my program says only 6 hours 23 minutes. This is most probably caused, by fact that I don't distinguish which day of a week it is. I didn't consider it, because for map, it is good enough to be just approximate.

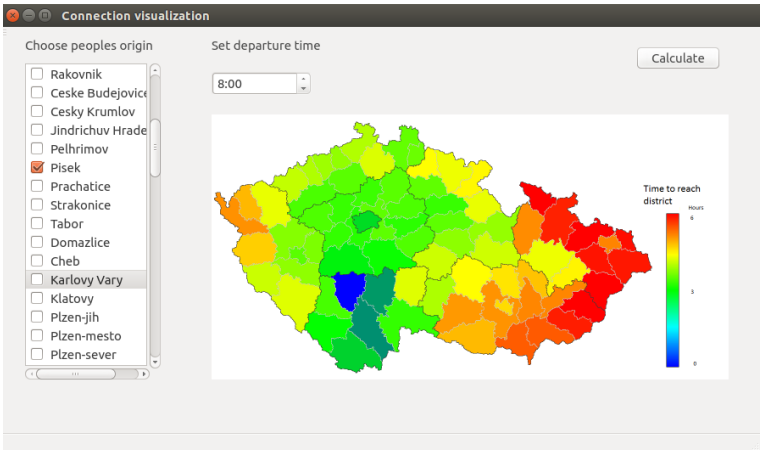
Last source will be Pisek. Connection to Pelhrimov will arrive at 12:10 according to IDOS. My program says 4 hours and 10 minutes. Sumperk - arrival 13:27, program agrees if travels on Sunday. Here are find there is Regiojet connection with invalid data format. Last Bruntal, IDOS 14:23, Program 6 hours 23 minutes. In the Figure 3.1 you can see all three heat maps.



(a) Praha as source



(b) Karlovy Vary as source



(c) Pisek as source

Figure 3.1: Heat-maps for three different sources

Table 3.1: Three sources calculation

Destination tier	Program output	IDOS answers
Pelhrimov	256 = 4:26	(160+250+360)/3 = 256 = 4:26
Sumperk	291 = 4:51	(206+341+327)/3 = 291 = 4:51
Bruntal	343 = 5:43	(263+473+383)/3 = 373 = 6:13

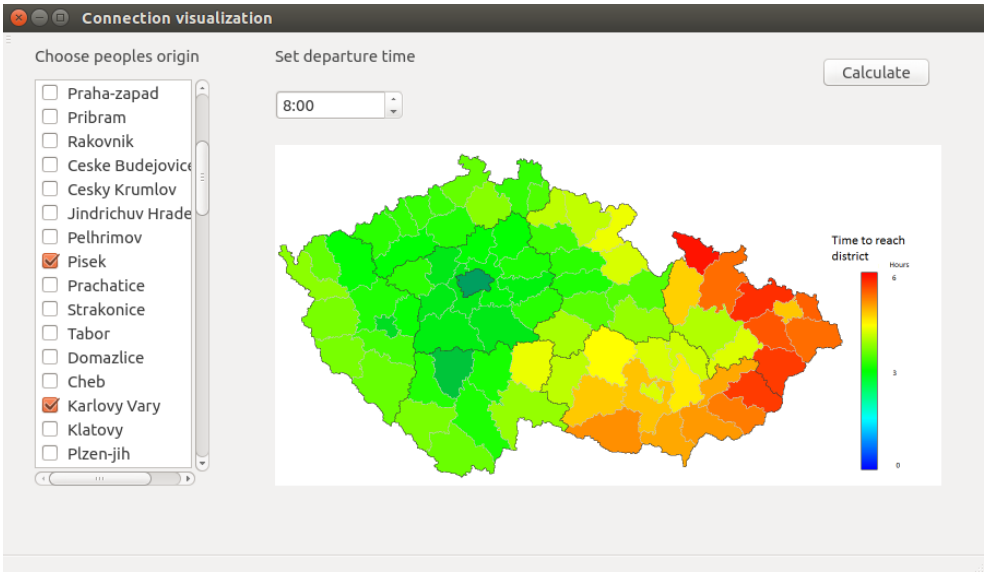


Figure 3.2: Heat map for three people from different locations

3.2 Three sources

Now I will test, if calculation for three people from different places also gives correct results. To have something to valid results with, our three source tiers will be Praha, Karlovy Vary and Pisek. Also we have to keep same destinations, because only this way, we will be able to validate results. In the *table* 3.1 you can see three columns. First one is name of selected destination, in second there is time value program returns for each destination. In the last one, there is calculation of three times for each destination, as calculated in previous section. We are calculating average time needed to reach that place. So that we add all three values together and then divide them by 3. As you can see, first two lines have same result for IDOS and our program. The difference in last one is caused by different time calculated between Karlovy Vary and Sumperk. Reasons for that were already mentioned above. In the Figure 3.2 you can see how the heat map looks for three people from different places.

As did we check above, only one connection was find differently in IDOS

compared to my program. It is quite good result, however not perfect. This difference is not really surprising for many reasons. As it was already mentioned, I didn't take into consideration day of a week, also not all carriers provide data in proper format. Another reason might be, imported schedules are already outdated, as unexpected changes are no provided through data.

Conclusion

My initial motivation why to chose this topic and try to solve the problem, was fact, public transport data for Czech Republic in machine-readable form were only released recently. More and more I was digging into it, more it was becoming obvious, things are not as easy, as they might look. Data were really published, but their formats are obscure. The formats being used by Chaps are worldwide unique. So that no additional information could be find. Furthermore format for trains is completely different from one for buses. Also specification differs with every new version. What's more, several informations are missing, mostly location of stops and changes in train schedules. Making it almost impossible to use for standard planner. After I somehow solve import of data, I focused on graph representation and shortest path problem. I inspected several algorithms, only to chose Dijkstra's algorithm as most suitable. As I had to do some modification, I learned perfectly how it works. In last part I was facing a problem of visualizing results I get. Because data were lacking GPS coordinates, almost all conventional solutions didn't suits me. So I had to find my own way, how to do so. Output of my thesis is desktop application, which is capable of showing best place to meet using GUI. Even with few complications, I am happy, I did choose this topic. During analysis and implementation I gained a lot of new knowledge about graphs, algorithms in them as well as about public transport planning and open-data.

Possibilities of further work

There is many possibilities how to extend my work. Because data source is main problem, it will be very useful to be able to automatically transform input format into worldwide spread GTFS. Once this will be done, one could use tons of already done solutions not only for finding fastest connection, but also to visualize these data etc. However this will be a lot of work, as we will have to add GPS coordinates to every stop manually and probably even more.

Question is if it will be useful as I still believe Chaps will be finally forced to provide data in GTFS format.

Bibliography

[1] Developers, G. GTFS Static Overview [online]. https://developers.google.com/transit/gtfs/, [Accessed: 2018-02-09].

[2] Wiki, T. Publicly-accessible public transportation data [online]. https://www.transitwiki.org/TransitWiki/index.php/Publicly-accessible_public_transportation_data, [Accessed: 2018-02-09].

[3] GTFS.ORG. GTFS Background [online]. http://gtfs.org/gtfs-background/, [Accessed: 2018-02-09].

[4] dopravy, M. Jízdní řády veřejné dopravy [online]. ftp://ftp.cisjr.cz/, [Accessed: 2018-02-09].

[5] Iyanaga, S.; Kawada, Y. *Encyclopedic Dictionary of Mathematics*. MIT press, 1980.

[6] Radek, S.; Roman, S.; Kominkova, O. Z.; et al. *Software Engineering Perspectives and Application in Intelligent Systems*. Springer, 2016.

[7] Wehrmeyer, S. Dynamic Public Transport Travel Time Map [online]. https://www.mapnificent.net/prague/, [Accessed: 2018-02-09].

[8] of Tranportation in Czech Republic, M. Popis JDF verze 1.10 [online]. https://www.chaps.cz/files/cis/jdf-1.10.pdf, [Accessed: 2018-02-09].

[9] of Tranportation in Czech Republic, M. Databaze kmenovych dat vlaku [online]. ftp://ftp.cisjr.cz/draha/celostatni/V%FDm%ECnn%E9%20soubory%20KANGO%202017-10-22.pdf, [Accessed: 2018-02-09].

[10] Qt. Qt GUI [online]. https://doc.qt.io/qt-5.10/qtgui-index.html, [Accessed: 2018-02-09].

[11] Noske, A. Code - heatmaps and color gradients [online]. http://www.andrewnoske.com/wiki/Code_-_heatmaps_and_color_gradients, [Accessed: 2018-02-09].

Appendix A

Acronyms

GUI Graphical user interface

GTFS General Transit Feed Specification

JDF Data format used by Chaps for buses

GPS Global Positioning System

OOP Object-oriented programming

API Application programming interface

Appendix B

Contents of enclosed CD

- `readme.txt` ..... the file with CD contents description
- `bp2` ..... executable program
- `Makefile` ..... makefile to make program
- `configure.sh` ..... bash script to prepare input files
- `src` ..... the directory of source codes
  - `thesis` ..... the directory of LaTeX source codes of the thesis
- `text` ..... the thesis text directory
  - `thesis.pdf` ..... the thesis text in PDF format
- `data` ..... the directory of input data
- `img` ..... the directory of images
- `documentation` ..... the directory of documentation